CHAPTER 9

# Perl and CGI Programming

**case** Dominion Consulting is offering a special promotion of software products. To give potential customers another method of responding to the promotion, the company's Sales Department asks you to create an interactive Web page. You can do so by creating scripts in Perl, a programming language similar to C that uses features from awk and shell programs.

LESSON A

objectives

**In this lesson you will:**

- **Learn the basics of the Perl language**
- **Create simple Perl scripts with variables and logic structures**
- **Create simple Perl scripts to read and sort data files**

# Learning to Use Perl

In this chapter, you learn how to use Perl to create effective, interactive Web pages. **Perl** (which stands for Practical Extraction and Report Language) was created in 1986 by Larry Wall as a simple report generator. Since then, the author and others have enhanced it so that it has become a powerful programming language. One of the most popular uses of Perl scripts today is to make interactive Web pages.

This chapter also expands on your previous knowledge of both awk and sed. You will learn more options and features of both by writing awk and sed scripts as problem-solving programs, not just as isolated commands within other programs.

Before you can begin to create your Web page, however, you first need to learn more about the structure and syntax of a Perl program.

## Introduction to Perl

Perl contains a blend of features found in other languages. It is very similar to the C language but also contains features found in awk and shell programs. You will begin learning Perl by examining a few simple programs, such as this one:

```
#!/usr/bin/perl
# Program name: example1.p
print("This is a simple\n");
print("Perl program.\n");
```

The first line in the program tells the operating system to use Perl to interpret the file. Recall from Chapter 7 that when the first line of a program begins with #!, the remainder of the line is assumed to give the path of the interpreter.

The second line in the sample program is a comment that lists the name of the file. Like shell scripts, Perl programs use the # character to mark the beginning of a comment. The third and fourth lines of the program display text on the screen. The program output is shown in Figure 9-1.

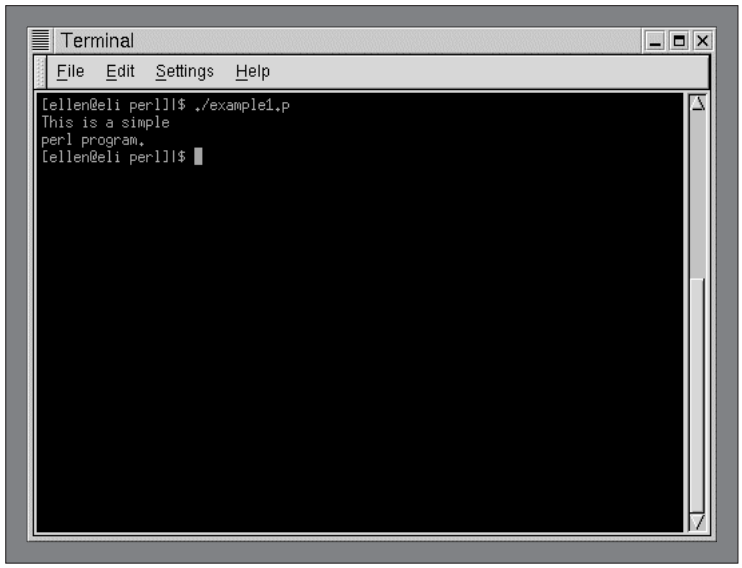



**Figure 9-1:** Output of example1.p

The print statements each have a single argument, which is displayed on the screen. The first print statement displays the string, "This is a simple." The \n characters display a new-line, which advances the cursor to the beginning of the next line. The second print statement is similar to the first. It displays the string, "Perl program," and then advances the cursor to the beginning of the next line. Notice that the two print statements end with a semicolon. All complete statements in Perl end with a semicolon.

> **Note: The parentheses surrounding the print statement's argument are optional. For example, these two statements perform the same operation:**

```
print ("Hello");
print "Hello";
```

Look at the next program, which uses a variable.

```
#!/usr/bin/perl
# Program name: example2.p
$name = "Charlie";
print("Greetings $name\n");
```

Figure 9-2 shows the program's output.

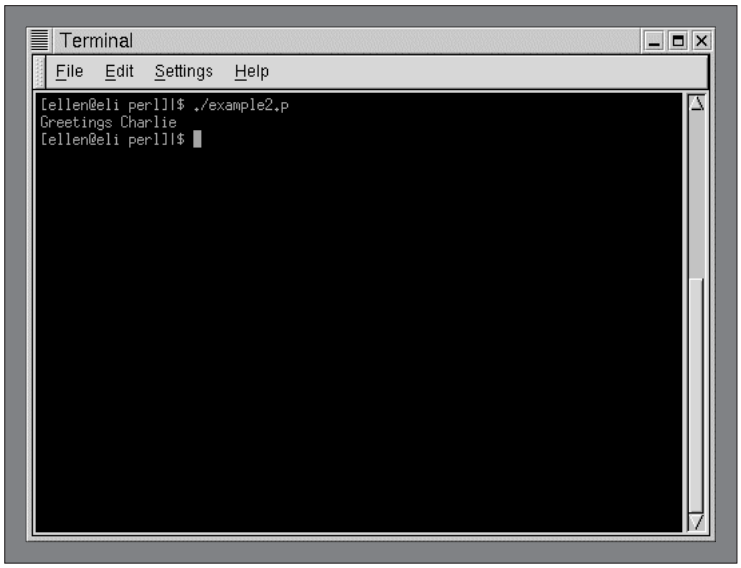



**Figure 9-2:** Output of example2.p

The example2.p program uses the variable $name. The variable is initialized with the string "Charlie." Notice that when $name is inserted in the print statement's argument, it displays the contents of the variable.

Perl can also read input from the keyboard. The next program is an example.

```
#!/usr/bin/perl
# Program name: example3.p
print ("Enter a number: ");
$number = <STDIN>;
print ("You entered $number\n");
```

The program's output is shown in Figure 9-3.

In Perl, <STDIN> reads input from the keyboard. The program uses this line to assign keyboard input to the variable $number:

```
$number = <STDIN>
```

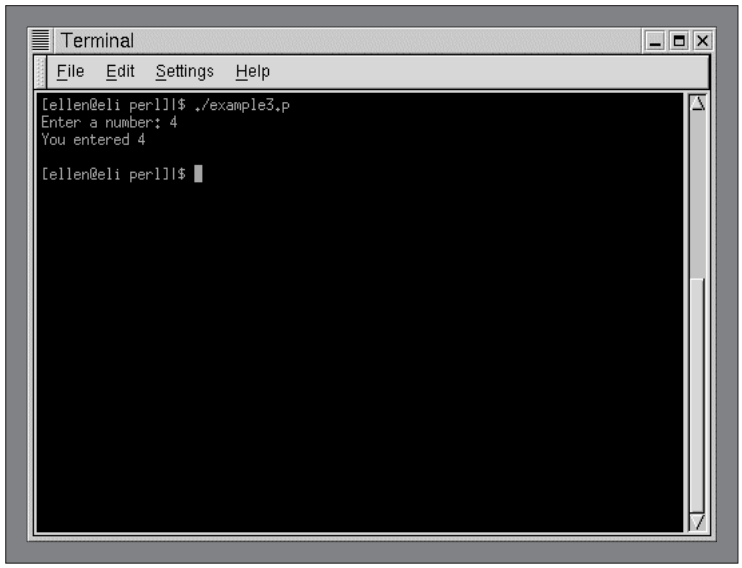



**Figure 9-3:** Output of example3.p

Like other languages, Perl offers the if-else statement as a decision structure. Here is an example:

```
#!/usr/bin/perl
# Program name: example4.p
print ("Enter a number: ");
$number = <STDIN>;
if ($number == 10)
{
     print ("That is the number I was thinking of.\n");
}
else
{
     print ("You entered $number\n");
}
```

The == operator tests two numeric values for equality. The if statement uses the == operator to determine if $number is equal to 10. If it is, the block (which consists of lines of code enclosed inside a set of curly braces) immediately following the if statement is executed. Otherwise, the block that follows the else statement is executed. Figure 9-4 shows the output of the program when the user enters 10.

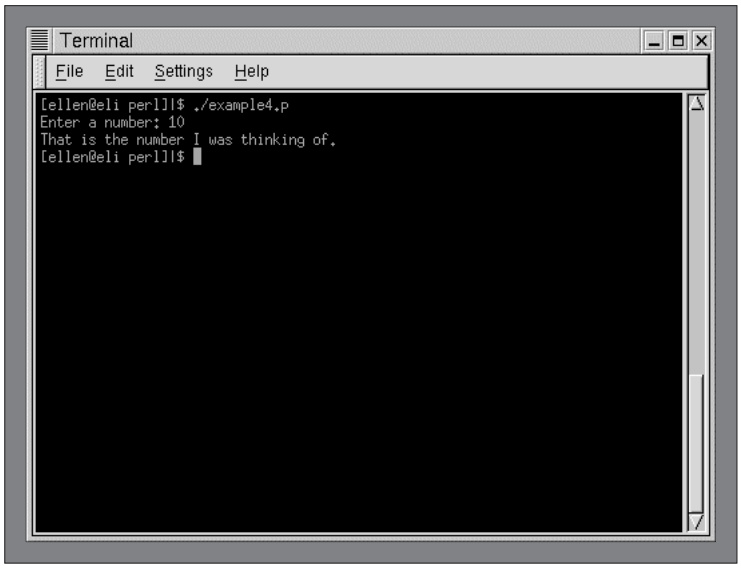



**Figure 9-4:** Output of example4.p when you enter 10

Figure 9-5 shows the output of the program when the user enters a value other than 10.

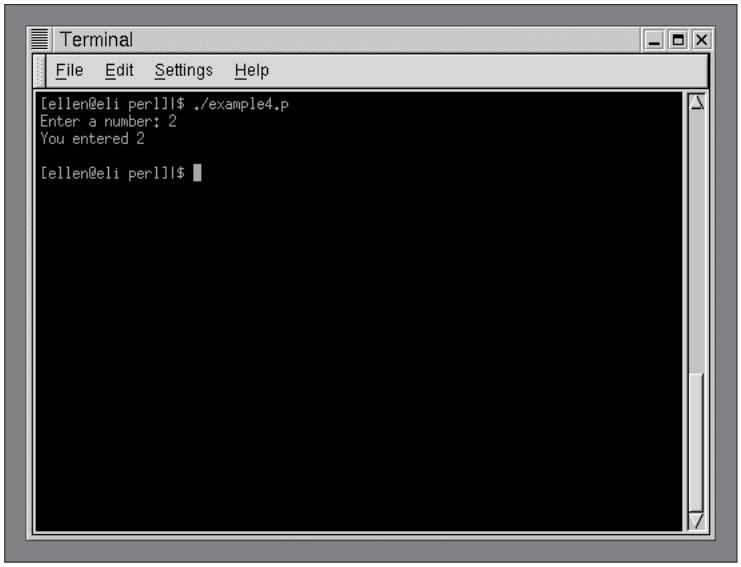



**Figure 9-5:** Output of example4.p when you enter a value other than 10

Perl also has operators that test for less than, greater than, less than or equal to, and greater than or equal to relationships. Table 9-1 shows each of Perl's numeric relational operators.

| Operator | Meaning |
|---|---|
| == | Equality |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| != | Not equal to |

**Table 9-1:** Perl's numeric relational operators

Perl can also perform relational tests on strings, which are sequences of characters. The string relational operators, however, are different from the numeric relational operators. The next program demonstrates how two strings, stored in variables, are compared for equality.

```
#!/usr/bin/perl
# Program name: example5.p
$my_name = "Ellen";
$your_name = "Charlie";
if ($my_name eq $your_name)
{
     print ("Your name is the same as mine.\n");
}
else
{
     print ("Hello. My name is $my_name\n");
}
```

The eq operator tests two strings to determine if they are equal. The output of the program is shown in Figure 9-6.

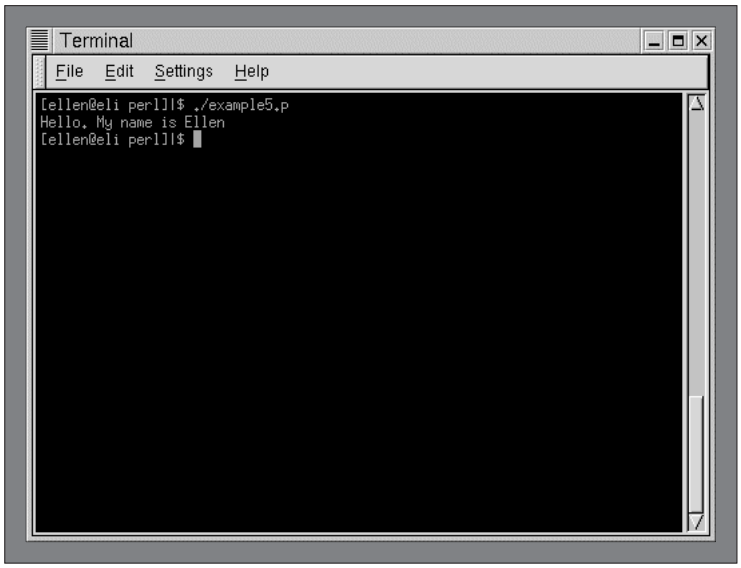



**Figure 9-6:** Output of example5.p

Table 9-2 lists Perl's string relational operators.

| Operator | Meaning |
|---|---|
| eq | Equality |
| lt | Less than |
| gt | Greater than |
| le | Less than or equal to |
| ge | Greater than or equal to |
| ne | Not equal to |

**Table 9-2:** Perl's string relational operators

Perl also provides standard arithmetic operators: + performs addition, - performs subtraction, * performs multiplication, and / performs division. The next program demonstrates a simple arithmetic operation.

```
#!/usr/bin/perl
# Program name: example6.p
$num1 = 10;
$num2 = 50;
$num3 = 12;
$average = ($num1 + $num2 + $num3) / 3;
print ("The average is $average\n");
```

Figure 9-7 shows the output of the program.

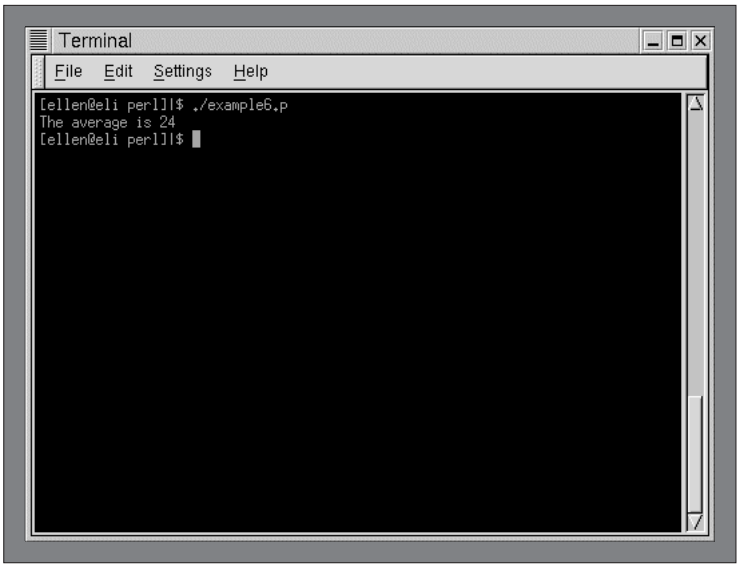



**Figure 9-7:** Output of example6.p

As you can see from the program above, Perl also lets you group operations within parentheses. Now that you have a general understanding of Perl, you will study its data types.

## Identifying Data Types

The computer programmer must understand not only what is contained in files, records, and fields, but also the format in which it is stored. Are the fields of information numeric or alphabetic? Are the fields made up of a combination of numbers and letters? How do you treat control characters such as tab and new-line? Although it may seem obvious that a data item such as a person's name cannot be added or multiplied, the programmer must write code that properly handles any and all data items that appear in a program. Otherwise, misidentified data generates processing errors. To do this, programmers need to identify data types.

Data may be represented in a Perl program in a variety of ways. You will learn about these types of data:

- Variables
- Constants
- Scalars
- Arrays
- Hashes

## Variables and Constants

**Variables** are symbolic names that represent values stored in memory. For example, the variable $x might hold the value 100, and $name might hold the sequence of characters, "Charlie." The value of a variable can change while the program runs. **Constants**, however, do not change value as the program runs. They are written into the program code itself. For example, this statement assigns the value of the constant 127.89 to the variable $num:

```
$num = 127.89
```

## Scalars

In the broadest sense, data is perceived as being either numeric or non-numeric. A non-numeric field of information is treated simply as a string of characters (hence the term *string*). Programmers associate strings with such items as a person's name, address, or license plate number. Numbers can also be used for logical analysis as well as for mathematical computations. A **scalar** is a simple variable that holds a number or a string. Scalar variable names begin with a dollar sign ($).

## Numbers

Numbers are stored inside the computer as either signed integers (as in 14321) or double-precision floating-point values (as in 23456.85). Numeric literals (constant values versus variable values) can be either integers or floating-point values. These numeric representations are consistent with all languages, but Perl also uses an additional convention with numeric literals to improve legibility: the underscore character, as in 5_456_678_901. (Perl uses the comma as a list separator.) The underscore only works within literal numbers specified in a program, not in strings functioning as numbers or in data read from elsewhere. Similarly, hexadecimal constants are expressed with the leading 0x prefix (as in 0xfff), and octal constants are expressed with the leading 0 prefix (as in 0256).

All of these are examples of statements that assign values to numeric scalar variables:

```
$x = 12;
$name = "Jill";
$pay = 12456.89;
```

## Strings

Strings are often used for logical analysis, sorts, or searches. **Strings** are sequences of any types of characters (including numbers that are treated as "characters" rather than digits). String literals are usually delimited by either single quotes (' ') or double quotes (" "). Single-quoted strings are not subject to interpolation (except for \' and \\, used to put single quotes and backslashes into a single-quoted string). Double quotes use the backslash (\) to precede a character that is to be treated as a control character. Table 9-3 lists the code and meaning for each string representation.

| Code | Meaning |
|---|---|
| \n | New-line |
| \r | Carriage Return |
| \t | Horizontal Tab |
| \f | Form Feed |
| \b | Backspace |
| \a | Bell |
| \033 | ESC in octal |
| \x7f | Del in hexadecimal |
| \cC | Ctrl-C |
| \\ | Backslash |
| \" | Double quote |
| \u | Force next character to uppercase |
| \l | Force next character to lowercase |
| \U | Force all following characters to uppercase until \E is encountered |
| \L | Force all following characters to lowercase until \E is encountered |
| \Q | Backslash—quote all following non-alphanumeric characters until \E is encountered |
| \E | End \U, \L, \Q |

**Table 9-3:** Double-quoted string representation

For example, compare the use of special codes in the next program with those shown in Table 9-3.

```
#!/usr/bin/perl
# Program name: example7.p
print ("\\words\\separated\\by\\slashes\n");
print ("This is a \"quote\"\n");
print ("\Uupper case\n");
print ("\LLOWER CASE\n");
```

The program output is shown in Figure 9-8.

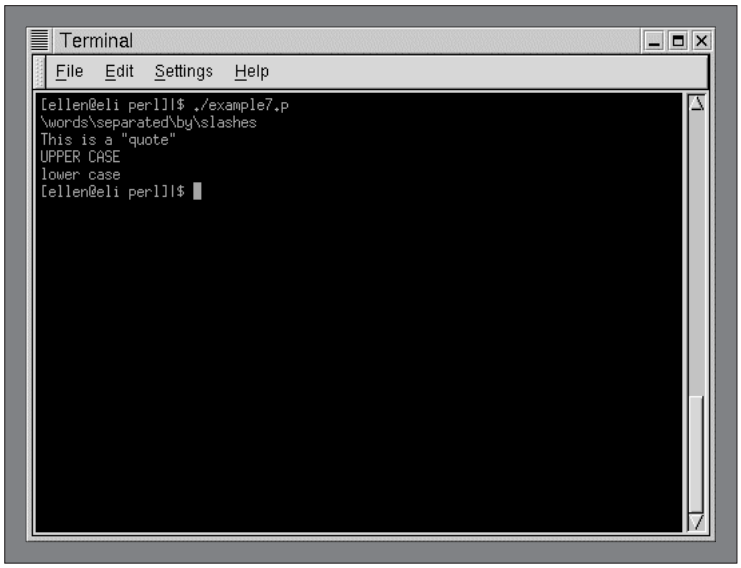



**Figure 9-8:** Output of example7.p

## Arrays

**Arrays** are variables that store an ordered list of scalar values that are accessed with numeric subscripts, starting at zero. An "at" sign (@) precedes the name of an array when assigning it values. When processing the individual elements of an array, however, use the $ character. For example, this program creates the array pets.

```
#!/usr/bin/perl
# Program name: example8.p
@pets = ("dog", "cat", "parrot", "hamster" );
print ("My pets are:\n");
print ("$pets[0]\n");
print ("$pets[1]\n");
print ("$pets[2]\n");
print ("$pets[3]\n");
```

Figure 9-9 shows the program output.

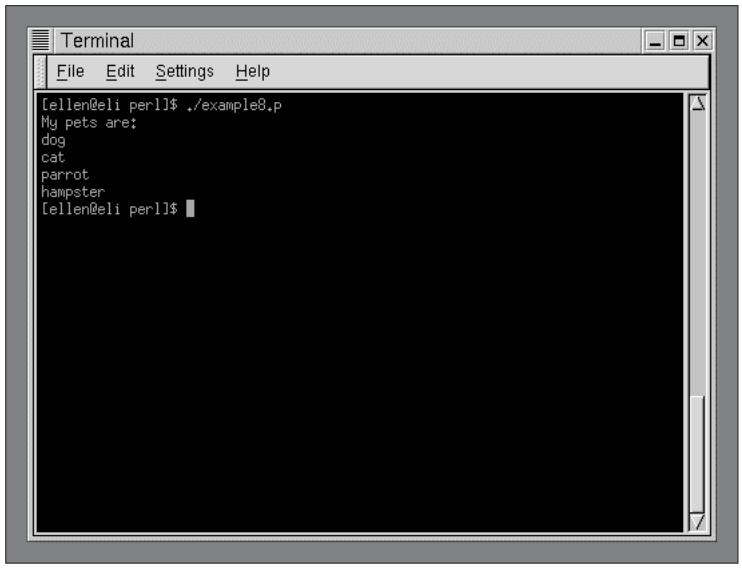



**Figure 9-9:** Output of example8.p

## Hashes

A **hash** is a variable that represents a set of key/value pairs. Hash variables are preceded by a percent sign (%) when they are assigned values. To refer to a single element of a hash, you use the hash variable name followed by the "key" associated with the value in curly braces. For example:

```
%animals = ('Tigers', 10, 'Lions', 20, 'Bears', 30);
$animals{'Bears'}
```

returns the value 30. Another, more readable way to define this is to use the ==> operator to define the key/value pairs:

```
%animals = (Tigers ==> 10, Lions ==> 20, Bears ==> 30);
```

Here is a program that demonstrates the use of a hash variable.

```
#!/usr/bin/perl
# Program name: example9.p
%animals = ('Tigers', 10, 'Lions', 20, 'Bears', 30);
print ("The animal values are:\n");
print ("$animals{'Tigers'}\n");
print ("$animals{'Lions'}\n");
print ("$animals{'Bears'}\n");
```

The program's output is shown in Figure 9-10.

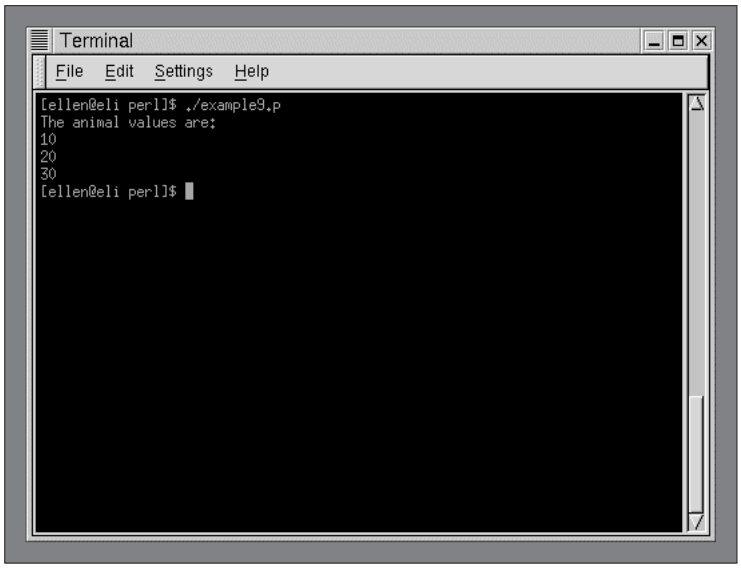



**Figure 9-10:** Output of example9.p

Now that you understand about data types, you are ready to learn more about programming using Perl. Perl's similarities and differences with other programming languages can be illustrated by comparing how the same program appears in awk and Perl.

## Perl versus Awk Programs

Awk does not require the programmer to explicitly set up looping structures as Perl does. Perl's while loop, on the other hand, is almost identical to the one found in C and C++. Awk, therefore, uses fewer lines of code to resolve pattern-matching extractions than Perl does. For example, look at the following awk program, awkcom.a, and its output. The program counts the number of comment lines that appear in the file specified on the command line.

```
#!/usr/bin/awk -f
# program name: awkcom.a
# purpose: Count the comment lines in a file
#          Enter the filename on the command line.

END {
     print "The file has ", line_count, " comment lines."
}
/^#/ && !/^#!/ { ++line_count }  # This occurs for every line.
```

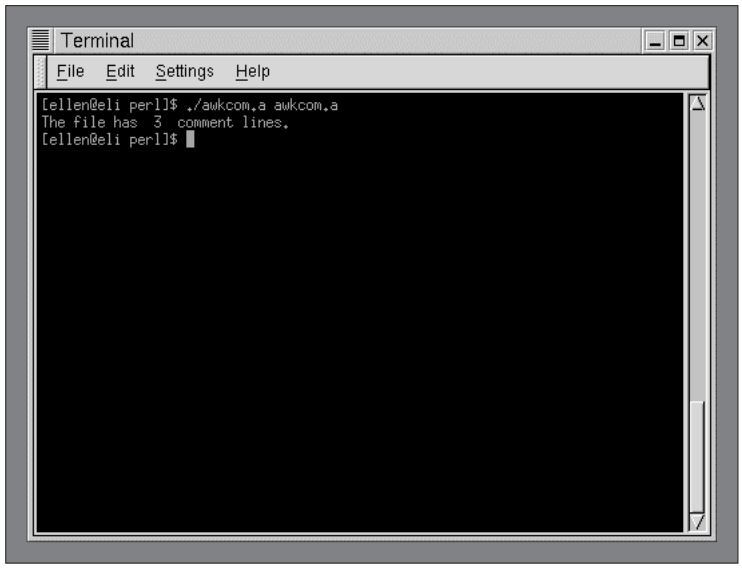



**Figure 9-11:** Output of awkcom.a

Now compare and contrast the awkcom.a program with this Perl program:

```
#!/usr/bin/perl
# program name: perlcom.p
# purpose: count the source file's comment lines

$filein = $ARGV[0];
while (<>)
{
     if (/^#/ && !/^#!/)
     {
          ++$line_count
     }
}
print ("File \"$filein\" has $line_count comment lines. \n"
);
```

Although the end results of both programs are very similar, you can see where the two programs differ. Awk uses an implicit while loop that automatically sends the entire contents of the file named on the command line to the pattern-matching and action part of the program. However, note that for the Perl program you need to build the while loop explicitly.

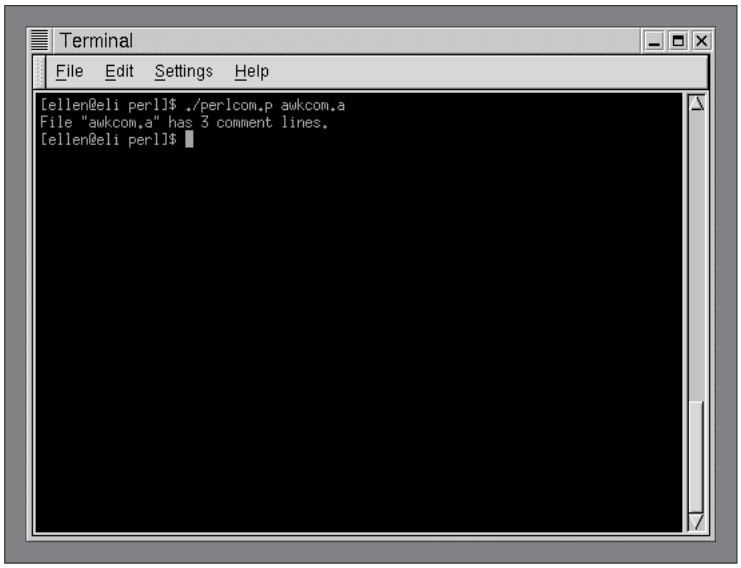



**Figure 9-12:** Output of perlcom.p

The first line of each program tells the shell program to run either the awk program or the Perl program and pass the statements in the file to the program for execution. Both programs also use the pound sign (#) to specify a comment line. Further, the pattern-matching code is the same in both programs. That is where the similarities end.

The –f option in the awk program tells the shell that the program is being called with a script file that contains the awk commands. Recall that awk contains more built-in commands to read lines from the file. All awk needs is the pattern-matching conditions to select the lines. The reading of the file, in awk, is implied as shown in this code.

```
/^#/&& !/^#!/ { ++line_count }
```

Awk also uses BEGIN and END to control when commands execute. All statements in a BEGIN block execute before the input file is read. All statements in an END block execute after all the contents of the input file have been read. This program only needs the END pattern.

In the Perl program, the code:

```
$filein = $ARGV[0];
```

takes the name of the file on the command line (ARGV[0]) and places it in a variable so it can be referenced later. The filename originally stored in ARGV[0] from the command line is destroyed during the while loop.

```
while (<>)
```

The <> symbol is called the **diamond operator.** Once the file is opened, you can access its data using the diamond operator. Each time it is called, it returns the next line from the file.

Curly braces open and close a block where you can place multiple statements:

```
if (/^#/ && !/^#!/)
{
     ++$line_count
}
```

This block tests to see if the line begins with the # character, but not with the #! characters. If true, the statement ++$line_count adds one to the $line_count variable and then closes the if block.

Whether awk or Perl is a good choice for you is a personal decision, but one should be part of your tool kit. There is no substitute for the kinds of work that either can perform very quickly, with minimal code preparation. For example, you probably would not want to write a C program for a task like scanning files for a matching pattern. Perl and awk are excellent when you are looking for a "needle in the haystack." Some say that the greater flexibility and power of Perl's expanded regular expressions give it a slight edge.

## How Perl Accesses Disk Files

Like most high-level programming languages, Perl uses filehandles to reference files. A **filehandle** is the name for an I/O connection between your Perl program and the operating system, and it can be used inside your program to open, read, write, and close the file. The convention is to use all uppercase letters for filehandles. In most instances, you must issue an open statement to open the file before you can access it. The exception to this occurs when you use the ARGV[0] variable to pass the filename to the program through the command line. In effect, you "open" it on the command line. As with other languages, every Perl program has three filehandles that are automatically opened: STDIN (the keyboard), STDOUT (the screen, to which the print and write functions are written by default), and STDERR (the screen, used to display error messages).

You will now learn common methods for opening and processing external files. The first program, perlread.p1, passes the filename on the command line, using the standard array variable that is reserved to do just that, ARGV[0]. This Perl program displays the contents of a file. (Recall that you can also use cat, less, and more for doing this.)

**To use Perl to display the contents of a file:**

1. Use the cat command (or the editor of your choice) to create the test file students, containing the names Joseph, Alice, Mary, Zona, Aaron, Barbara, and Larry.
2. Save the file.

3 Use the editor of your choice to create the Perl program perlread.p1:

```
#!/usr/bin/perl
# program name: perlread.p1
# purpose: Display records in a file and count lines
$filein = $ARGV[0];
while (<>)
{
     print "$_";
     ++$line_count
}
print ("File \"$filein\" has $line_count lines. \n");
```

4 Save the file and quit the editor.

5 Give the file execute permission.

6 Test the program by typing **./perlread.p1 students** and then pressing **Enter**. Your screen should now display the contents of the students file, shown in Figure 9-13.

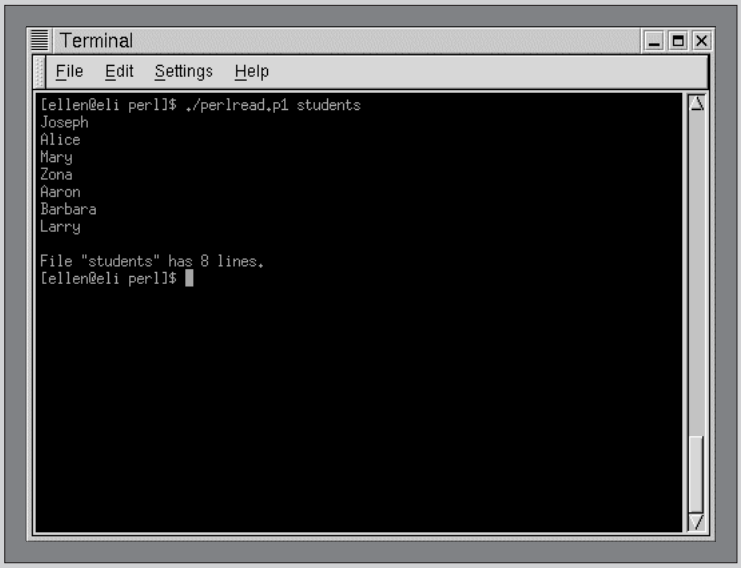



**Figure 9-13:** Output of perlread.p1

The first instruction ($filein = $ARGV[0];) saves the name of the file that is passed to the program and stores it in ARGV[0]. The while loop triggers the diamond operator (<>) that sequentially reads records from the file and places the value stored in ARGV[0] in the next record. This continues until the loop reaches the end of the file. When that happens, ARGV[0] contains a null (end-of-file character), so you cannot use ARGV[0] to reference the filename when the while loop

terminates. Two commands inside the while loop are enclosed within curly braces: print "$_" displays each record that is read in, and ++$line_count increments (counts) the records in the file. The last command, print ("File \$filein\", has $line_count lines. \n") prints the name of the file (saved in $filein) and the number of lines in the file.

Next, you learn how to open the file from within your program, as opposed to passing it on the command line. All files opened inside programs must be closed before the program terminates.

**To use Perl to open a file from within a program:**

1. Use the editor of your choice to create the file perlread.p2.
2. Enter this Perl program:

```
#!/usr/bin/perl
# program name: perlread.p2
# purpose: Open disk file. Read and display the records in
#          the file. Count the number of records in the
#          file.

open (FILEIN, "students") || warn "Could not open students
file\n";
while (<FILEIN>)
{
     print "$_";
     ++$line_count;
}
print ("File \"students\" has $line_count lines. \n");
close (FILEIN);
```

3. Save the file and quit the editor.
4. Give the file execute permission.
5. Test the program by typing **./perlread.p2** and then pressing **Enter**. Your screen should then display the contents of the students file, shown in Figure 9-14.

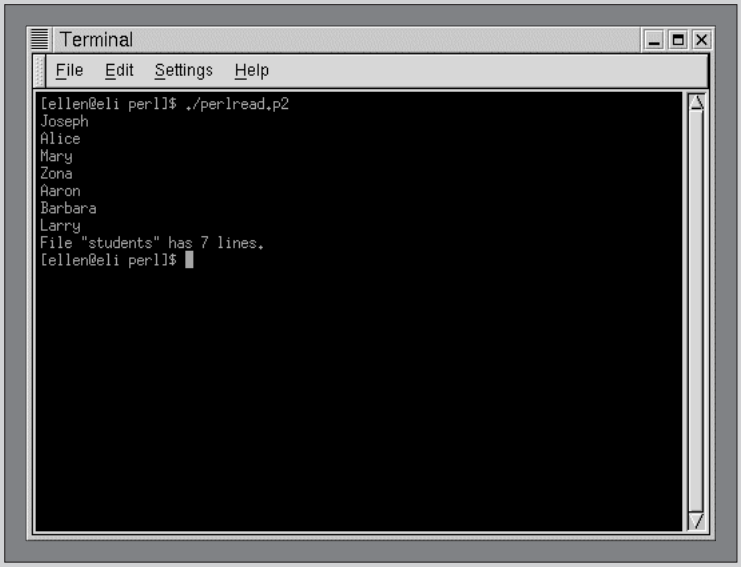



**Figure 9-14:** Output of perlread.p2

In the perlread.p2 program, the open function appears on line 5:

```
open (FILEIN, "students") || warn "Could not open students
file\n";
```

Nearly all program functions are written to return a value that indicates whether the function was carried out successfully. The values returned are considered true or false. A **true value** is usually represented with a 1, and sometimes any value greater than zero. A **false value** is represented with a 0 (zero). The open function returns true if the file is opened successfully and false if it failed to open. Opening a file can fail because the file is not found or because the file's permissions for reading and/or writing are not set. However, in Perl, a filehandle that has not been successfully opened can still be read, but you will get an immediate **EOF** (end-of-file signal), with no other noticeable effects. An EOF results in your program not letting you read from or write to the file, because the file is not available.

The two pipe characters "||" are the logical OR operator. When an expression on the left of a logical OR operator returns false, the expression on the right of the operator executes. The warn operator, on the right of the OR operator, displays an error message indicating the file did not open. Although displaying error conditions is not absolutely necessary in your programs, you should display them when it is obvious that the errors will cause subsequent problems if the program continues to run. This additional coding is especially essential in open statements.

After the file is open, access to the data is made through the diamond operator (<FILEIN>). When the diamond operator reaches the end of the file, it terminates the while loop. Except for the open and close statements and the use of the diamond operator, the perlread.p2 program is identical to perlread.p1.

## Using Perl to Sort

One of the most important tasks in managing data is organizing it into a useable format. Perl provides a powerful and flexible sort operator. It can sort string or numeric data in ascending or descending order. It even allows advanced sorting operations where you define your own sorting routine.

### Using Perl to Sort Alphanumeric Fields

You will now sort words in a Perl program into alphabetical order using the sort function.

**To use Perl's sort function:**

1. Use the editor of your choice to create the program perlsort.p1. Enter the code:

```
#!/usr/bin/perl
# program name: perlsort.p1
# purpose: Sort a list of names contained inside an array
# Syntax: perlsort.p1 <Enter>
#============================================================
@somelist = ( "Oranges", "Apples", "Tangerines",
              "Pears", "Bananas", "Pineapples"
            );
@sortedlist = sort @somelist;
print "@sortedlist";
print"\n";
```

2. Save the file and exit the editor.
3. Use the chmod command to grant the file execute permission.
4. Run perlsort.p1. Your screen should look similar to Figure 9-15.

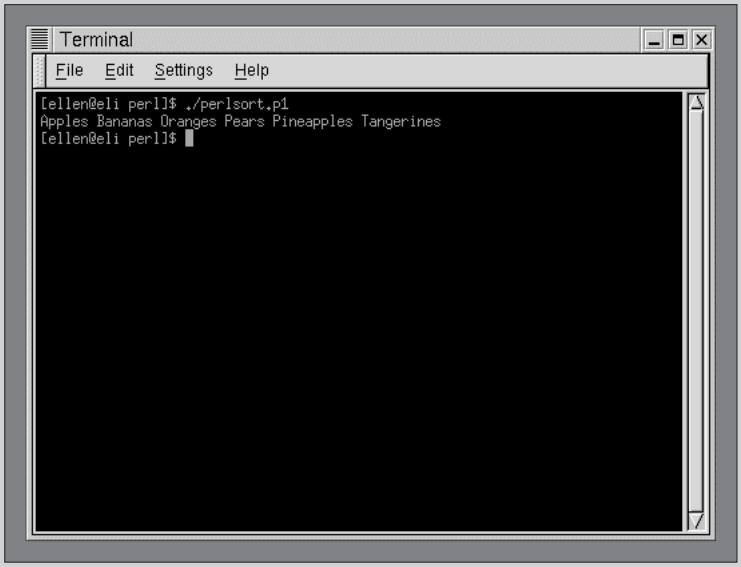



**Figure 9-15:** Output of perlsort.p1

Looking at the program, the statement:

```
@somelist = ( "Oranges", "Apples", "Tangerines",  "Pears",
"Bananas", "Pineapples");
```

puts the value of (Oranges, Apples, Tangerines, Pears, Bananas, Pineapples) into @somelist. The statement:

```
@sortedlist = sort @somelist;
```

calls the Perl sort function and returns the sorted output to the array variable, @sortedlist. The last two statements in the program print the sorted results and skip a line before the program terminates and returns to the command line.

Data is not always coded as part of the program or entered at the keyboard. Often, programs must read information from files. The next example demonstrates how Perl accesses a file by passing the filename on the command line.

**To use Perl to access a file by passing the filename on the command line:**

1. Use the editor of your choice to create the program perlsort.p2. Enter the code:

```
#!/usr/bin/perl
# program name: perlsort.p2
# purpose: Sorts a text file alphabetically. Filename is
#          entered on the command line.
# Syntax: perlsort.p2 filename <Enter>
```

```
#============================================================
$x = 0;
while (<>)
{
  $somelist[$x] = $_;
  $x++;
}
@sortedlist = sort @somelist;
print @sortedlist;
```

2. Save the file and exit the editor.
3. Give the perlsort.p2 file execute permissions.
4. Run perlsort.p2, using students as the test file, by typing **./perlsort.p2 students** and then pressing **Enter**. Your screen should now display the list of student names, shown in Figure 9-16.

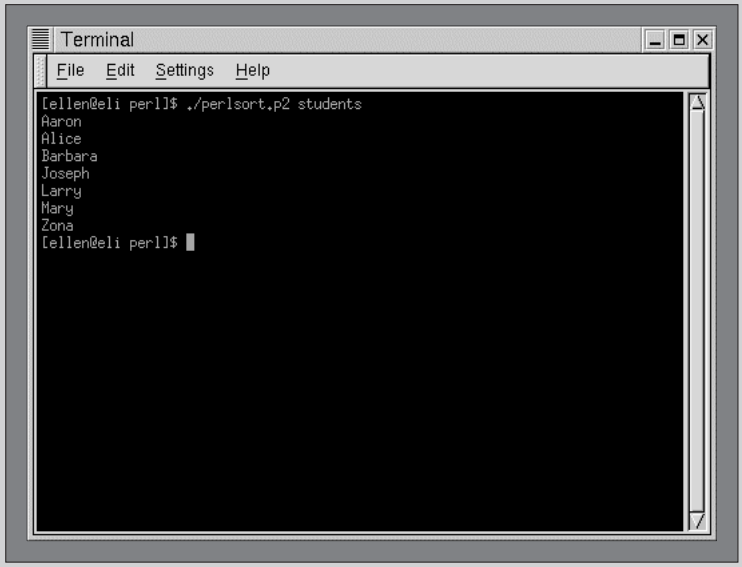



**Figure 9-16:** Output of perlsort.p2

The perlsort.p2 program uses the statement:

```
$x = 0;
```

to initialize a variable, $x, to contain an index to the array. The first element of every array is zero (0). In the while loop,

```
while (<>)
{
   $somelist[$x] = $_;
   $x++;
```

the next line in the file is automatically copied into the $_ variable. The assignment statement:

```
   $somelist[$x] = $_;
```

copies the contents of the $_ variable into an element of the array. The element is determined by the variable $x, which is used as a subscript. After the assignment operation occurs, the following statement executes:

```
   $x++;
```

The ++ operator adds one to its argument, so the statement increments the variable $x. As a result, the first name, Aaron, is placed in $somelist[0], Alice is placed in $somelist[1], and so on.

The statement:

```
@sortedlist = sort @somelist;
```

sorts the array, @somelist, placing the alphabetized names into @sortedlist, and the final instruction prints the alphabetized list of students' names.

## Using Perl to Sort Numeric Fields

Sorting numeric fields requires using a subroutine where you can define comparison conditions (e.g., greater than, less then, or equal to) between the data you are sorting. The sort routine is then called repeatedly, passing two elements to be compared on each call. The scalar variables $a and $b store the two values that are compared to select the larger value. Using the comparison operation, a return code of –1, 0, or +1 is returned, depending on whether $a is less than, equal to, or greater than $b, as in the demonstrated code:

```
sub numbers
{
   if ($a < $b) { -1; }
   elsif ($a == $b) { 0; }
   else { +1; }
}
```

When sorting numbers, you need to instruct Perl to use this sort subroutine as the comparison function, rather than the built-in ASCII ascending sort (the default). To do this, place the name of the subroutine between the keyword *sort* and the list of items to be sorted:

```
   $sortednumbers = sort numbers 101, 87, 34, 12, 1, 76;
```

The statement instructs Perl to sort the values in the list but use the numbers subroutine to determine their order. The output is in numeric order, not ASCII order.

The numeric comparison of $a and $b is performed so frequently that Larry Wall, Perl's creator, developed a special Perl operator for numeric sorts, <=>. This sort operator, known as the **spaceship operator**, reduces coding requirements. To illustrate the code savings, compare the next sort subroutine using the spaceship operator with the previous one:

```
sub numbers
{
     $a <=> $b;
}
```

This numbers subroutine produces the same result as the first example, which uses an if-else statement. Perl allows an even more compact notation: the **inline sort block**, which looks like this:

```
@sortednumbers = sort { $a <=> $b; } @numberlist;
```

This statement uses the block { $a <=> $b; } as the sort routine. It eliminates the need for a separate subroutine. Let's examine how a Perl program sorts numeric data.

**To use Perl for numeric sorting:**

1. Create the file numberlist, containing the data **130**, **100**, **121**, **101**, **120**, and **122**.
2. Use the editor of your choice to create the perlsort.p3 program. Enter this code:

```
#!/usr/bin/perl
# program name: perlsort.p3
# purpose: Sorts numerically using a subroutine. Filename
#          is entered on the command line.
# Syntax: perlsort.p3 filename <Enter>
#============================================================
$x = 0;
while (<>)
{
     $somelist[$x] = $_;
     $x++;
}
@sortedlist = sort numbers @somelist;
print @sortedlist;

sub numbers
{
     if ($a < $b)
          {-1; }
     elsif ($a == $b)
          { 0; }
     else
          {+1; }
}
```

3. Save the file and exit the editor.
4. Use the chmod command to grant the file execute permission.
5. Test the program by typing **./Perlsort.p3 numberlist** and then press **Enter**. Your screen should appear similar to Figure 9-17.

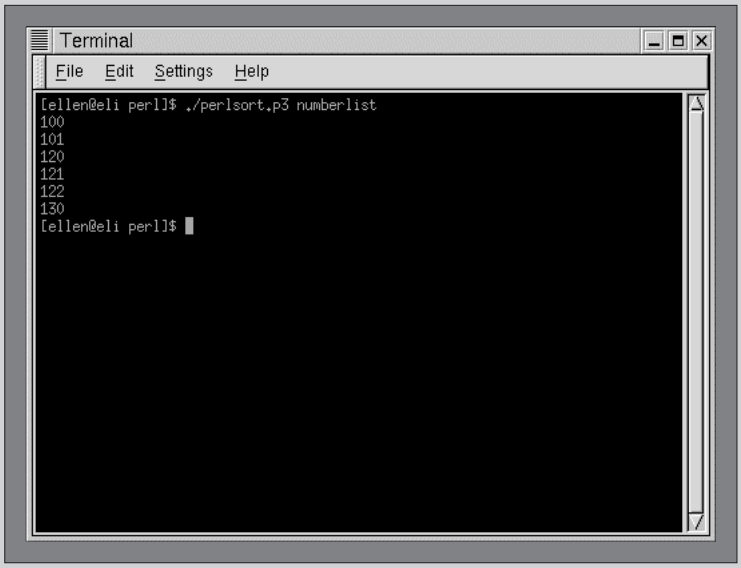



**Figure 9-17:** Output of perlsort.p3

The perlsort.p3 program uses a sort subroutine that compares $a and $b numerically rather than textually and initializes the array element index to start with the first element, 0. The while loop,

```
while (<>)
{
     $somelist[$x] = $_;
     $x++;
}
```

works the same as previously described, in that it reads records from a file and stores the lines inside an array.

The sort subroutine,

```
sub numbers
{
     if ($a < $b) { -1; }
     elsif ($a == $b) { 0; }
     else ( +1; }
}
```

compares the two numbers that are sequentially passed to it from the while loop. If the value in $a is less than the value in $b, the subroutine returns –1. If $a is equal to $b, the subroutine returns 0. Otherwise, the subroutine returns +1.

You will now see how using the spaceship operator can save you coding time.

**To use Perl's spaceship operator:**

1. Use the editor of your choice to create the program perlsort.p4. Enter this code.

```
#!/usr/bin/perl
# program name: perlsort.p4
# purpose: Sort numerically using spaceship operator (<=>)
# syntax: perlsort.p4 filename <Enter>
#=============================================================
$x = 0;
while (<>)
{
     $somelist[$x] = $_;
     $x++;
}
@sortedlist = sort numbers @somelist;
print @sortedlist;

sub numbers
{
     $a <=> $b;
}
```

2. Save the file and exit the editor.
3. Use the chmod command to grant the file execute permission.
4. Test the program by typing **./perlsort.p4 numberlist** and then press **Enter**. Again, your screen should display the list of numbers sorted in ascending order, as shown in Figure 9-18.

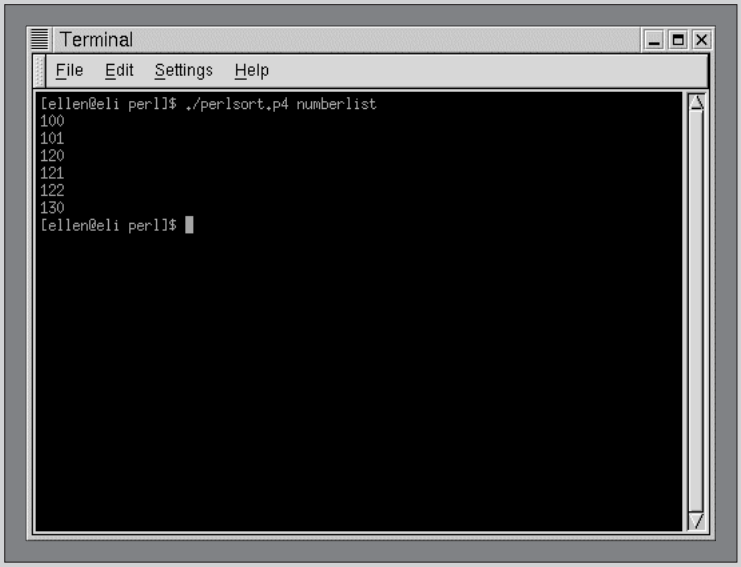



**Figure 9-18:** Output of perlsort.p4

In the perlsort.p4 program, notice that the only code changes to the perlsort.p3 program are those found in the shortened subroutine.

Now that you are more familiar with Perl, you will learn how to create a Web page. Then you will be ready to begin creating the Web page for Dominion Consulting.

## SUMMARY

- Perl is being extensively used as a powerful text-manipulation tool similar to awk.
- Perl is written in scripts that are translated and executed by the Perl program.
- The programmer has to write process handling instructions for data items to prevent misidentification of data types and subsequent processing errors.
- Perl has three basic data types: scalars, arrays, and hashes. A scalar is a simple variable, such as a number or a name. Scalar variable names begin with $. Arrays are ordered lists of scalars that are accessed with numeric subscripts, staring at zero [0]. Array variable names are preceded with the at sign, @. Hashes are unordered sets of key/value pairs that you can access using the keys as subscripts. Hash variables begin with the percent sign, %.

- A list is an ordered group of simple variables or literals, separated by commas. For example, (101, 102, 103, 104) is an array of four values, 101 through 104.
- Anything besides a textual sort must be handled with a sort subroutine for which you can provide your own comparison function to determine greater-than, less-than, or equal-to conditions between the elements being sorted.

## REVIEW QUESTIONS

1. To read data from a file, Perl _____.
   a. works just like awk
   b. uses a while loop like C programs
   c. uses the spaceship operator
   d. always uses an array to store the records from the file
2. Perl scripts _____.
   a. begin with line indicating that /usr/bin/perl is the interpreter
   b. are not made executable
   c. do not support the use of if statements
   d. do not support the use of while loops
3. The spaceship operator refers to _____.
   a. CGI programming
   b. a shortcut for sorting names in Perl
   c. a shortcut for sorting numbers in Perl
   d. a Perl in-line sort
4. What does the statement $x = <STDIN>; perform?
   a. displays the contents of $x
   b. copies the string "STDIN" to $x
   c. copies the contents of the variable STDIN to $x
   d. reads keyboard input into $x
5. The == operator _____.
   a. tests two numbers for equality
   b. tests two strings for equality
   c. tests either numbers or strings for equality
   d. performs an assignment
6. The eq operator _____.
   a. tests two numbers for equality
   b. tests two strings for equality
   c. tests either numbers or strings for equality
   d. performs an assignment

7. The $ character precedes _____.
   a. array names
   b. scalar variable names
   c. hash names
   d. constants
8. The @ character precedes _____.
   a. array names
   b. scalar variable names
   c. hash names
   d. constants
9. The % character precedes _____.
   a. array names
   b. scalar variable names
   c. hash names
   d. constants
10. Assume that a program contains the code:

```
@food = ("fruit", "steak", "bread", "vegetables" );
```

What does the next statement print in the same program?

```
print ("$food[2]");
```

   a. fruit
   b. steak
   c. bread
   d. vegetables

11. Assume that this code exists in a program:

```
%food = ("fruit", 5, "steak", 10, "bread", 15, "vegetables",
20 );
```

What does the next statement print in the same program?

```
print ("$food{'bread'}");
```

   a. 5
   b. 15
   c. steak
   d. bread

## EXERCISES

1. Write a Perl script to print “Hello Perl”.
2. Write a Perl script to sort the numbers 1, 8, 15, 1000, 12, which are located in a memory array.
3. Create the file Ex2numbers using the numbers from Exercise 2. Write a Perl script using the spaceship operator to sort and display Ex2numbers.

4. Write a Perl script with a hash variable. The hash variable should contain these names and telephone numbers:

| | |
|---|---|
| Jean James | 555-9898 |
| Rhonda Smith | 555-0982 |
| Joe Milner | 555-8944 |
| Greg Jones | 555-0716 |

The program should display the phone numbers of each individual.

## DISCOVERY EXERCISES

1. Modify the program perlcom.p (developed earlier in this chapter), so it counts all lines in the file that are not comments.
2. Write a Perl program to count the number of records in the students file (created earlier in this chapter) that begin with the letter A.
3. Write a Perl program that converts a value in inches to a value in centimeters and displays the result. (1 inch = 2.54 centimeters.)
4. Write a Perl program that uses a while loop to display the values 1 through 12 and their squares.
5. Write a Perl program that asks the user to enter two numeric values. Store the values in $x and $y. If $y is not zero, divide $x by $y and display the result. If $y is zero, display an error message indicating that division by zero is not possible.

LESSON B

objectives

**In this lesson you will:**

- **Create an HTML document for the World Wide Web**
- **Use Perl and CGI scripts to make your Web pages interactive**
- **Use X-Window and Netscape to retrieve Web pages**

# Creating an Interactive Web Page

## Setting Up a Web Page

You can create a Web page using **HTML (Hyper Text Markup Language)**. HTML is a format for creating documents with embedded codes known as **tags**. When the document is viewed in a Web browser, such as Netscape Navigator or Internet Explorer, the tags give the document special properties. Examples of properties include foreground and background colors, font size and color, and the placement of graphic images. In addition, HTML tags let you place **hyperlinks** in a document. A hyperlink is text or an object that, when clicked, loads another document and displays it in the browser.

After you use HTML to create a Web page, you then publish the page on a Web server. A **Web server** is a system connected to the Internet running Web server software, such as Apache. The Web server software lets other users access the HTML document via the Internet.

You may experiment with and test HTML documents using a UNIX or Linux system's localhost networking feature. The **localhost** feature allows your UNIX or Linux system to access its own internal network configuration instead of an external network. To use the localhost, you do not need to be connected to the Internet. More importantly, the localhost can emulate a real-world Web site, so you can carry out the testing and development of your new Web pages. Stand-alone testing of new Web pages is recommended: after fully testing your work, you can then later transfer your documents to any Web server, knowing that they are ready to perform.


**To run the Web pages and CGI programs in this lesson, you must have the Apache Web Server program installed on your Linux Computer. (See Appendix C for instructions on installing Apache.)**

## Creating Web Pages

You may use a visual HTML editor, such as Netscape Composer or Microsoft FrontPage, to create Web pages. These programs let you graphically construct a Web page in a "what you see is what you get" fashion. If you have no visual HTML editor, all you need is a text editor. You create the HTML document by typing its text and the desired embedded tags. Here is a sample HTML file.

```
<HTML>
<HEAD><TITLE>My Simple Web Page</TITLE></HEAD>
<BODY>
<H1>Just a Simple Web Page</H1>
This is a Web page with no frills!
</BODY>
</HTML>
```

All special codes contained inside angled brackets <> are tags. The first tag, <HTML>, identifies the file as an HTML document. Notice the corresponding </HTML> tag at the end of the file. Everything between the <HTML> and </HTML> tags are considered text with HTML tags. In general, most tags are used this way. One tag marks the beginning of a section, while a corresponding tag marks the end of the section.

Note that there are two parts to the code: a head and a body. The **head** contains the title, which appears on the top bar of your browser window. The **body** defines what appears within the browser window. All other tags refine the Web page's appearance. Figure 9-19 shows the Web page's appearance in the Netscape Web browser.

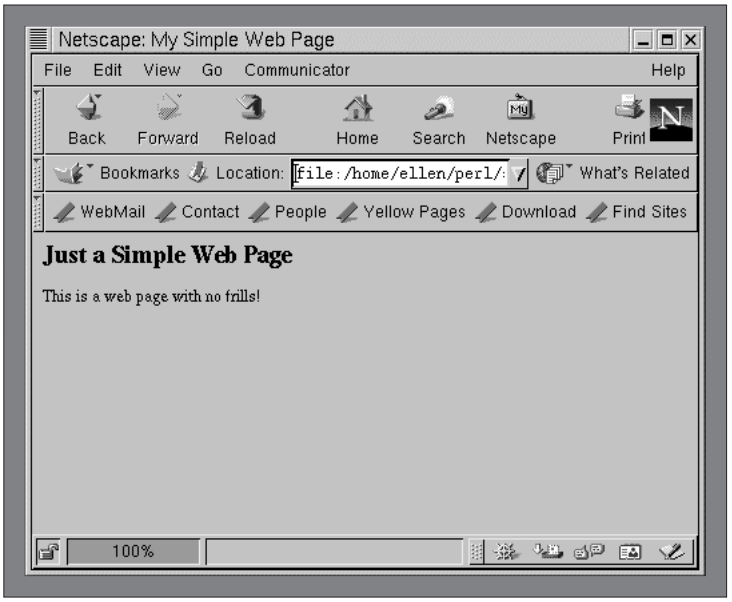



**Figure 9-19:** Simple Web page

You can use tags to set background and foreground colors and to manipulate text with such tags as <FONT SIZE=n>(*insert text here*). You can change text sizes with the heading tags, where <H1> is the largest and <H6> is the smallest. (However, note that users' browsers may also automatically change the actual text size.)

Because standard HTML ignores multiple spaces, tabs, and carriage returns, you can enclose text within <PRE></PRE>(preformatted text) tag pairs. Otherwise, any consecutive spaces, tabs, carriage returns, or combinations produce a single space. You can also use the <P> tag, which creates two line breaks, or the <BR> tag, which creates one line break. Neither tag requires a closing tag.

Browsers automatically wrap text so you don't need to worry about page widths. To center text, however, use <CENTER>(*text here*)</CENTER>. To indent from both margins, use <BLOCKQUOTE>(*text here*)</BLOCKQUOTE>. To change color, use <FONT COLOR="RGB"(*text here*)</FONT>, where RGB is the RGB color code. An **RGB color code** is a set of three numbers that specify a color's red, green, and blue components. For example, the code 512218 specifies a red component of 51, a green component of 22, and a blue component of 18. The higher the number, the more intense the color component.

Here is another example of an HTML file.

```
<HTML>
<HEAD><TITLE>UNIX Programming Tools</TITLE></HEAD>
<BODY>
<H1><CENTER>My UNIX Programming Tools</CENTER></H1>
<H2>Languages</H2>
<P>Perl</P>
<P>Shell Scripts</P>
<P>C and C++</P>
<H2>Editors</H2>
<P>vi</P>
<P>Emacs</P>
<H2>Other Tools</H2>
<P>awk</P>
<P>sed</P>
</BODY>
</HTML>
```

Figure 9-20 shows the file as it appears in a Web browser.

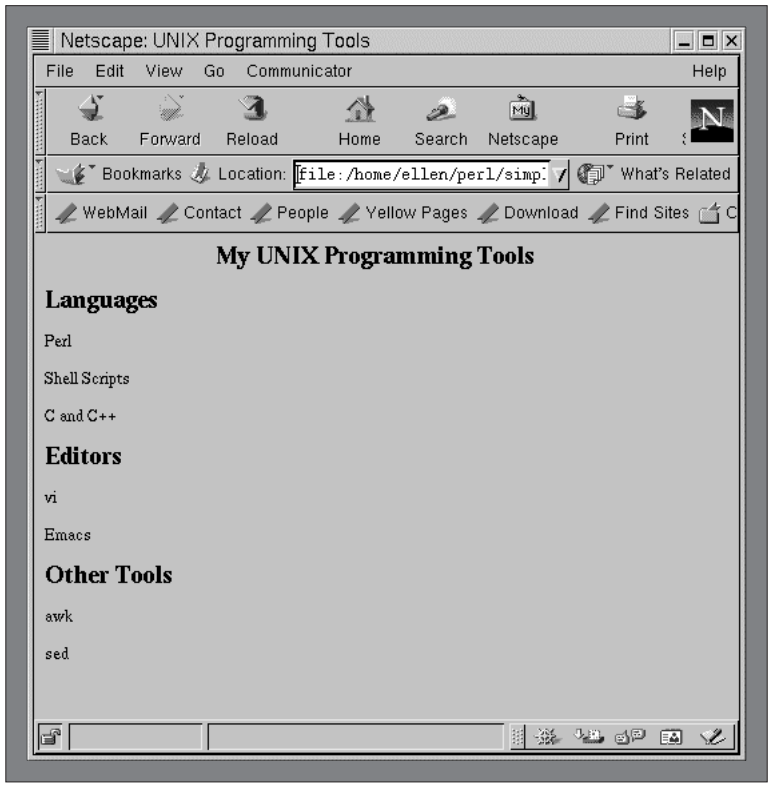



**Figure 9-20:** UNIX programming tools Web page

Now that you have general knowledge of creating Web pages, you need to learn how to use Perl and CGI to make them interactive.

## CGI Overview

Perl is the most commonly used language for **CGI (Common Gateway Interface) programming.** CGI is a protocol, or set of rules, governing how browsers and servers communicate. Any script that sends or receives information from a server needs to follow the standards specified by CGI. Thus, scripts written in Perl follow the CGI protocol. CGI Perl scripts are specifically written to get, process, and return information through your Web pages, that is, they make your Web pages interactive.

To allow your HTML document to accept input, especially where CGI rules apply, precede the input area with a description of what you want users to enter. For example, if you want users to enter cost, you would use this code:

```
Total Cost? <INPUT TYPE=text NAME=cost SIZE=10>
```

In addition, consistent with transmitting information to and from Web sites, you can use the special code INPUT TYPE=submit, which sends the data out when a user clicks the Submit button. The destination that you wish to receive the submitted

information is coded into the FORM tag. The **FORM tag** specifies how to obtain the information to be transferred. There are two methods, GET and POST. The **GET** method transfers the data within the URL itself. **POST** uses the body portion of the HTTP request to pass parameters. (You will use the POST method in this chapter.)

You can download hundreds of already written scripts and use them in your own Web page applications. Some of these scripts are free, such as subparseform.lib, which is used in these demonstrations. It was downloaded from http://www.cookwood.com/perl.

Other sources for free Perl CGI scripts are:

- http://www.worldwidemart.com/scripts
- http://www.extropia.com
- http://www.awsd.com/scripts

These sites provide useful Perl information and answers to FAQs (frequently asked questions):

- http://www.perl.com (a huge site that is a home to a vast collection of information)
- http://language.perl.com/faq/index.html
- http://language.perl.com/info/documentation.html

Before creating the Web page for Dominion Consulting, you will first see how a sample Web page works in UNIX.

**To see a sample Web page:**

1. To make a subdirectory to store your HTML, CGI scripts, and Perl scripts, type **cd /home/httpd/cgi-bin**, press **Enter**, type **mkdir <your user name>**, and then press **Enter**. For example, if you are the user Ellen, you will create the directory /home/httpd/cgi-bin/ellen.

   **Note: Make sure that the /home/httpd/cgi-bin directory is given full access permissions (usually done by the System Administrator). To do this on your own PC, log on as superuser, type chmod 777 /home/httpd/cgi-bin and then press Enter.**

2. Change your working directory to the new directory you created in Step 1.
3. See your instructor or technical support person for instructions for copying the following programs and scripts to the new directory:

   projest.html

   projest.cgi

   subparseform.lib
4. Start the X-Window system (the UNIX equivalent of Microsoft Windows) to run the Netscape browser. To do so, type **startx** and then press **Enter**. Your screen should look similar to Figure 9-21.

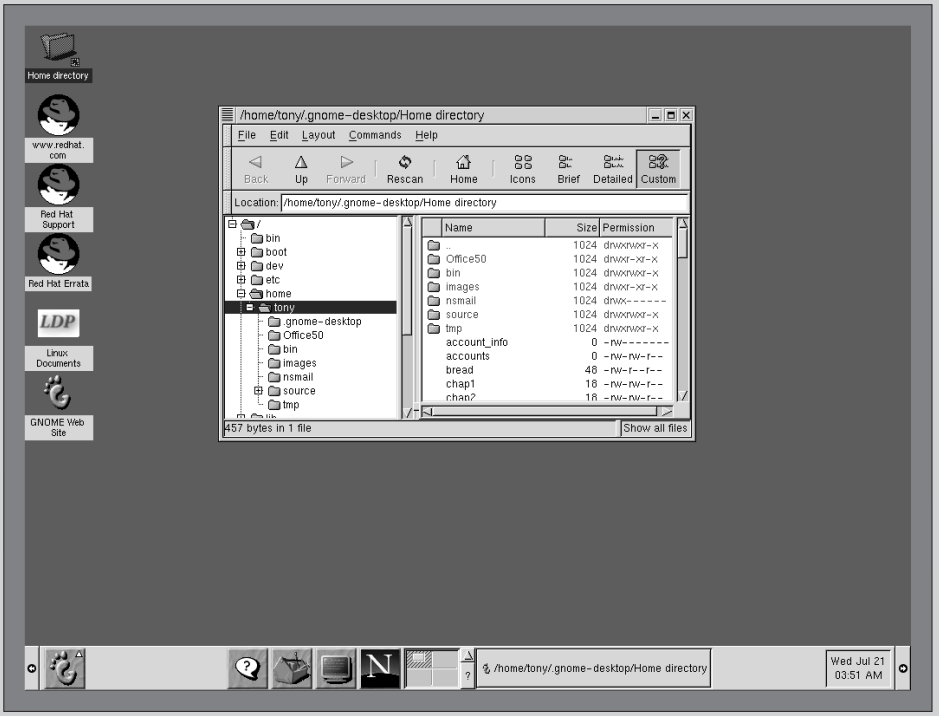



**Figure 9-21:** X-Window system

**5** Within X-Window, you can execute Netscape from a command line. You can access the command line in a Terminal window. To open a Terminal window, click the **Terminal** icon, illustrated in Figure 9-22.

**Figure 9-22:** Terminal icon

**6** The Terminal window appears, as shown in Figure 9-23. Click inside the Terminal window to make it active.

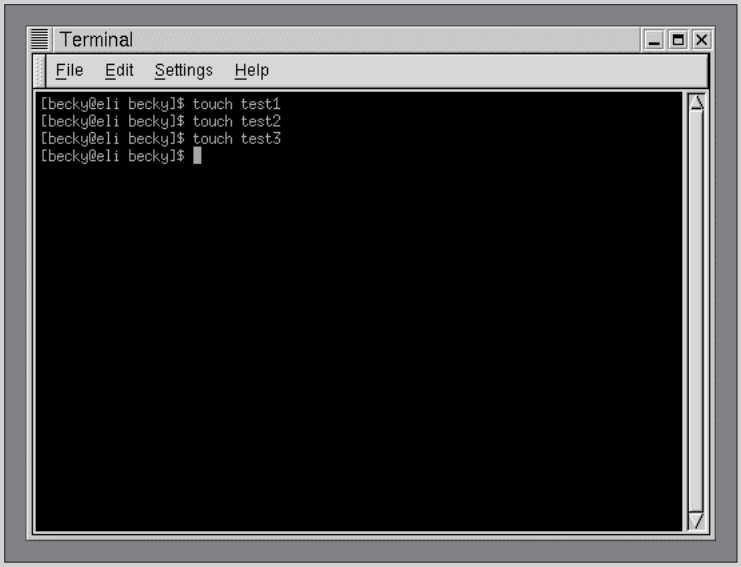



**Figure 9-23:** Terminal window

**7** You will view the file projest.html in the web browser, but first you must modify the file so it will know where its CGI script is located. Load the file projest.html into the editor of your choice. The contents of the file are shown below.

```
<!- Program Name: projest.html ->

<HTML><HEAD><TITLE>Dominion Project
Analysis</TITLE></HEAD>
<BODY>

<H2>Average Profit per Project Calculation</H@>
<FORM METHOD=POST ACTION="http://localhost/cgi-
  bin/ellen/projest.cgi">
Total cost of projects last year? <INPUT TYPE=text
NAME=projcost SIZE=10>
Number of Projects? <INPUT TYPE=text NAME=projects
  SIZE=10>
Project revenue received? <INPUT TYPE=text NAME=revenue
  SIZE=10>
<HR><INPUT TYPE=submit NAME=submit VALUE=Submit>
<INPUT TYPE=reset NAME=reset VALUE="Start over">
</FORM></BODY></HTML>
```

The line in boldface must be modified. Change the name "ellen" to your logon name. For example, if your logon name is sam, you will modify the line to read:

```
<FORM METHOD=POST ACTION="http://localhost/cgi-
bin/sam/projest.cgi">
```

Save the file and exit the editor.

**8** You must now execute the Netscape browser to see the sample Web page. To open Netscape Communicator, type netscape and then press Enter. The Netscape opening screen appears, as shown in Figure 9-24.

**Note: If this is the first time that you have accessed Netscape, you may have to click the Accept button at the lower right of the opening screen.**

**9** Click in the **Location** text box, press **Del** to delete the current text, and then type **file:/home/httpd/cgi-bin/<your user name>/projest.html** as the location of your Web page. For example, if you are the user Ellen, type file:/home/httpd/cgi-bin/ellen/projest.html.

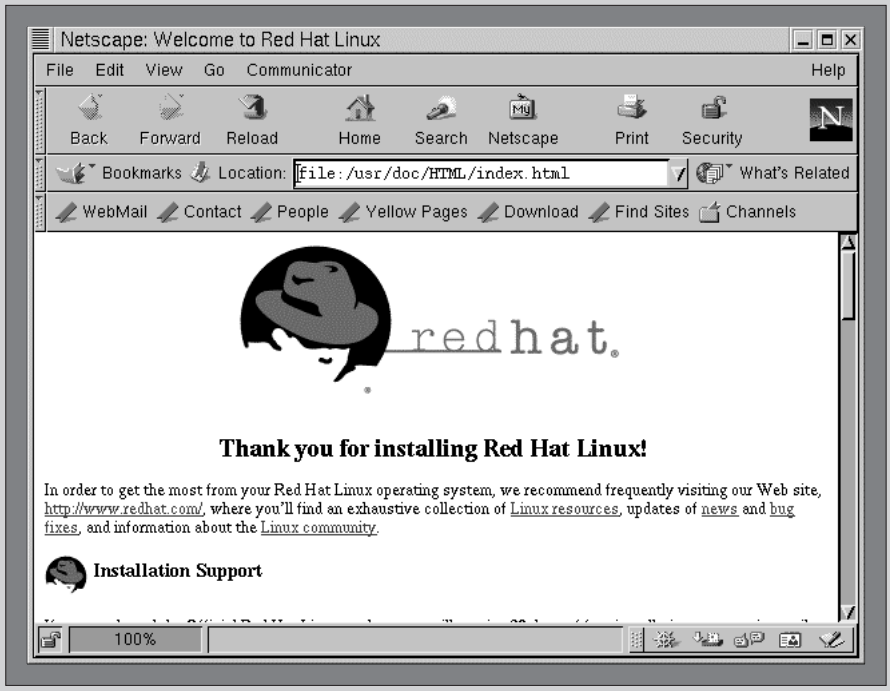
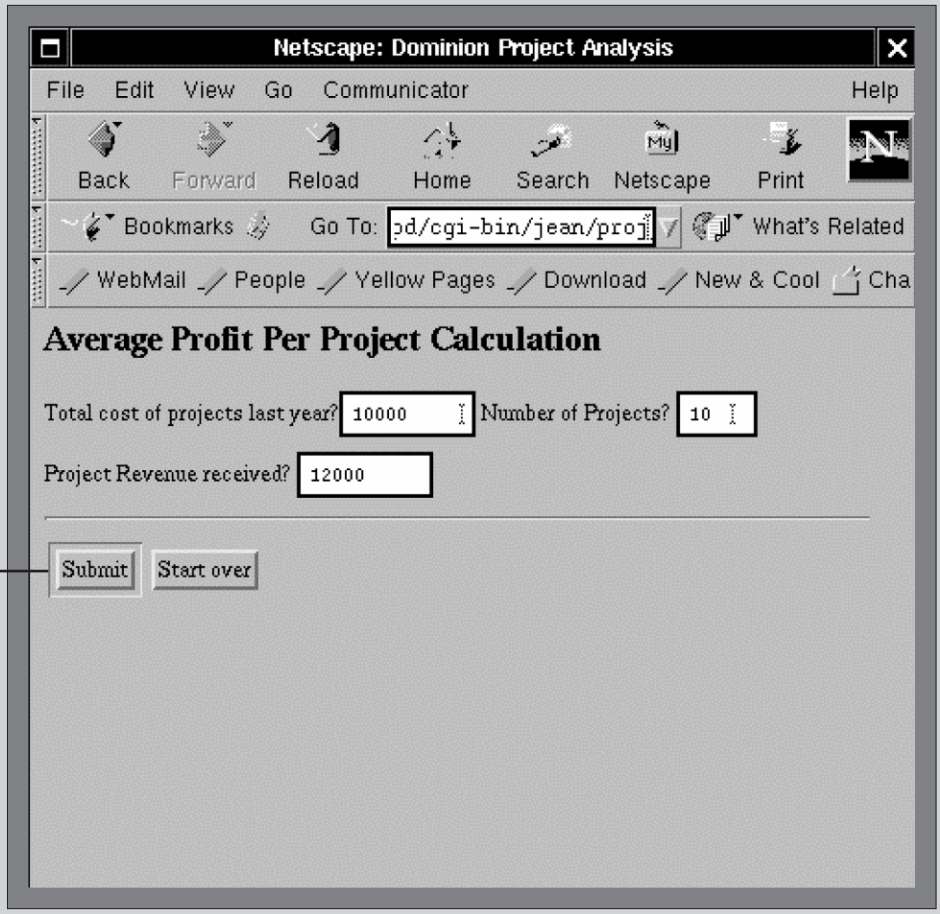



**Figure 9-24:** Netscape opening screen

**10** Press **Enter**. The opening screen for the projest.html Web page appears, as illustrated in Figure 9-25.

Here is the source code for the projest.html file.




**Click on the Submit button to ask Apache to use the Common Gateway Interface to retrieve and run the projest.cgi script**

**Figure 9-25:** Web page generated by projest.html

**11** To confirm that you want to use the Common Gateway Interface connection, click the **Submit** button in the Confirmation screen. (You can avoid the Confirmation screen in the future if you click the Show the Alert Next Time button.) Perl executes the program and the Apache server then passes the Web page response back to the Netscape browser for display. Your screen, the final screen that is part of the Perl/cgi script response, should now look similar to Figure 9-26.

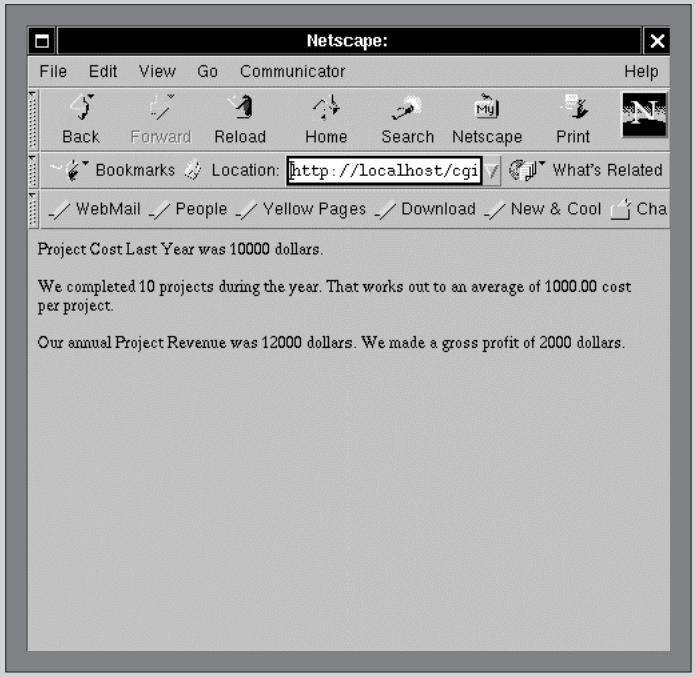



**Figure 9-26:** Final Perl/CGI script response screen

Here is the source code used to generate this Web page.

```
#!/usr/bin/perl
# Program name: projest.cgi
require "subparseform.lib;"

&Parse_Form;
$projcost = $formdata{'projcost'};
$projects = $formdata{'projects'};
$revenue = $formdata{'revenue'};

$average = $projcost / $projects;
$average = sprintf("%.2f", $average);
$grossprofit = $revenue - $projcost;

print "Content-type: text/html\n\n";
print "<P>Project Cost Last Year was $projcost dollars.";
print "<P>We completed $projects projects during the year.
That works out to an average of $average cost per
project.";
print "<P>Our annual Project Revenue was $revenue dollars.
We made a gross profit of $grossprofit dollars";
```

Now that you have seen a demonstration of how Web pages work using UNIX, you are ready to create your own Web page.

## Creating the Dominion Consulting Web Page

Dominion Consulting is currently offering all its hotel management customers a special promotional price for three customized applications. The company wants to present a Web page that offers customers an opportunity to order the promotional items over the Internet. The planned Web page is shown in Figure 9-27. You will now create that page.

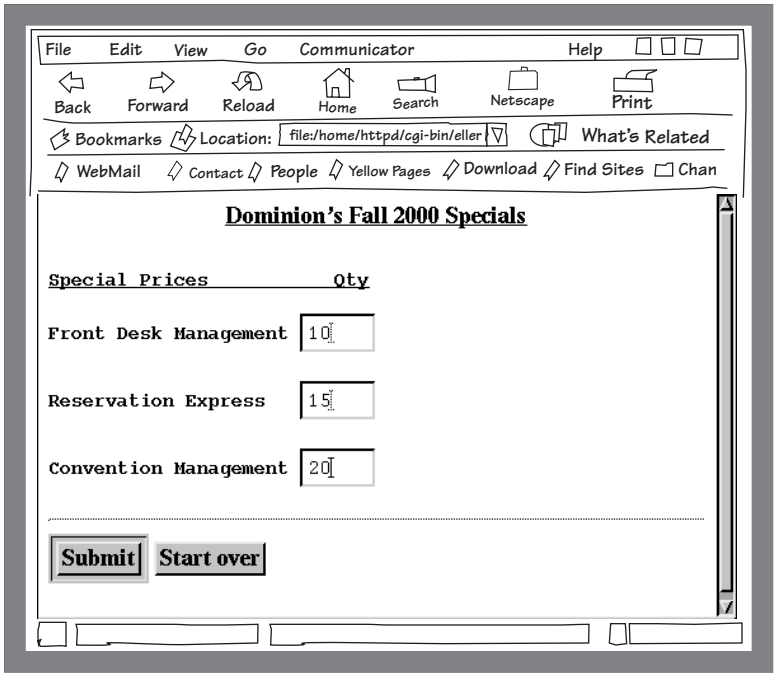



**Figure 9-27:** Planned Dominion Consulting sales promotion Web page

**To create a Web page:**

1. Use the editor of your choice to create the HTML document software.html. Enter this HTML code:

```
<!- Program Name: software.html->
<HTML><HEAD><TITLE>Dominion Consulting</TITLE></HEAD>
<BODY BGCOLOR=WHITE>
<CENTER><H1><U>Dominion's Fall 2000 Specials</H1></CENTER>
<FORM METHOD=POST ACTION="http://localhost/cgi-bin/<your
user name>/software.cgi">
<BR>
<H2><U><PRE>Special Prices          Qty</PRE></U></H2>
<FONT SIZE=5>
<PRE>Front Desk Management <INPUT TYPE=text NAME=frontdk
Size=5></PRE>
<PRE>Reservation Express   <INPUT TYPE=text NAME=reserve
```

```
SIZE=5></PRE>
<PRE>Convention Management <INPUT TYPE=text NAME=convmgt
SIZE=5></PRE>
<HR><INPUT TYPE=submit NAME=submit VALUE=Submit>
<INPUT TYPE=reset NAME=reset VALUE="Start over">
</FORM></BODY></HTML>
```

2. Save the file in your cgi-bin/<your user name> directory, and exit the editor.
3. Now use the editor to create the CGI Perl script software.cgi. Enter this code:

```
#!/usr/bin/perl
# Program name: software.cgi

require "subparseform.lib";&Parse_Form;
$frontdk = $formdata{'frontdk'};
$reserve = $formdata{'reserve'};
$convmgt = $formdata{'convmgt'};

$qtotal = $frontdk+$reserve+$convmgt;
$tfrontdk = $frontdk*200;
$treserve = $reserve*150;
$tconvmgt = $convmgt*180;
$total = $tfrontdk+$treserve+$tconvmgt;

print "Content-type: text/html\n\n";
print "";
print "</FONT><FONT  SIZE=6 PTSIZE=20>

</P><P ALIGN=CENTER><B><CENTER>Dominion Special</CENTER>

</P><P ALIGN=LEFT></FONT><FONT  SIZE=3 PTSIZE=10>
";
print "<CENTER></FONT><FONT  SIZE=5 PTSIZE=16>

<U>
Thank you for your order.</U>

</FONT><FONT  SIZE=3 PTSIZE=10></CENTER>";
print "
";
print "<TABLE BORDER=1 BGCOLOR=CYAN ALIGN=CENTER
WIDTH=300 CELLSPACING=5>";
print "<TR><TH ALIGN=CENTER>Qty</TH>";
print "<TH ALIGN=CENTER>Software</TH>";
print "<TH ALIGN=CENTER>Total</TH></TR>";
print "<TR><TD ALIGN=CENTER>$frontdk</TD>";
print "<TD ALIGN=CENTER>Front Desk Management</TD>";
print "<TD ALIGN=CENTER>\$$tfrontdk</TD></TR>";
print "<TR><TD ALIGN=CENTER>$reserve</TD>";
print "<TD ALIGN=CENTER>Reservation Express</TD>";
print "<TD ALIGN=CENTER>\$$treserve</TD></TR>";
print "<TR><TD ALIGN=CENTER>$convmgt</TD>";
print "<TD ALIGN=CENTER>Convention Management</TD>";
```

```
print "<TD ALIGN=CENTER>\$$tconvmgt</TD></TR>";
print "<TR><TD ALIGN=CENTER>$qtotal</TD>";
print "<TD ALIGN=CENTER>Total:</TD>";
print "<TD ALIGN=CENTER>\$$total</TD></TR></TABLE>";
</XMP></FONT><FONT  COLOR="#0f0f0f" BACK="#fffffe"
SIZE=3 PTSIZE=10>
```

4. Save the file in your cgi-bin/<your user name> directory, and exit the editor.
5. Use the chmod command to grant the file execute permission.

Now that you have entered both the code for the Web page and CGI script, you should test your work. Recall that with Linux Red Hat 6.0, you can use the localhost to carry out the testing and development of your new Web page.

**To test a Web page:**

1. Within X-Window, start Netscape Communicator by opening a Terminal window, typing **netscape** at the command line, and then pressing **Enter**.
2. In the location box, type **file:/home/httpd/cgi-bin/<your user name>/software.html**. Press **Enter.**
3. Enter quantities of **10, 15,** and **20** for the products, and then click the **Submit** button. Your screen should now look similar to Figure 9-28.

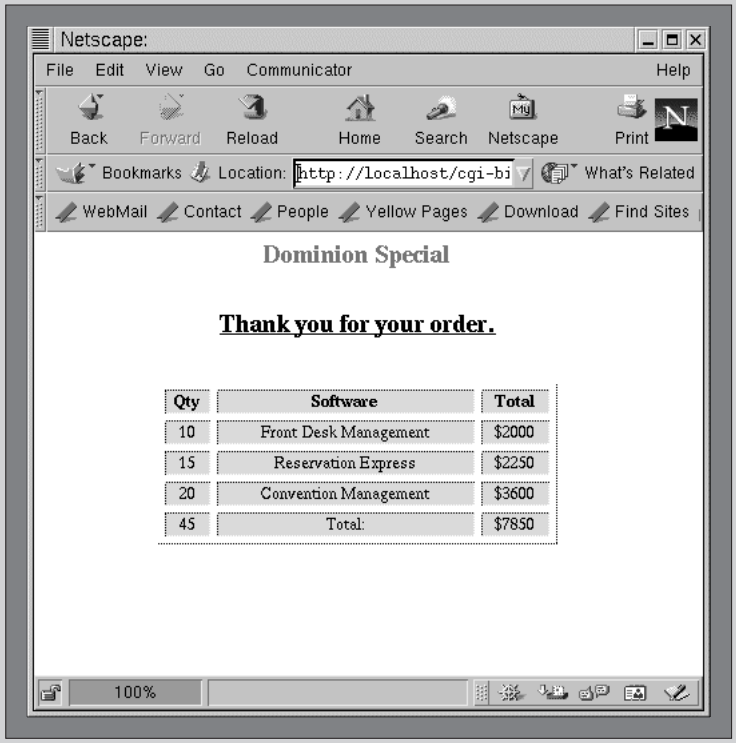



**Figure 9-28:** Web page returned to the browser from software.cgi

You have successfully created the Web page for Dominion Consulting by using your knowledge of Perl programming.

## SUMMARY

- An HTML document contains two parts: a head and a body. The head contains the title, which appears on the top bar of your browser window. The body defines what appears within the browser window.
- CGI (Common Gateway Interface) is a protocol or set of rules governing how browsers and servers communicate. Any script that sends or receives information from a server needs to follow the standards specified by CGI.
- To run your Web pages, you need to be in X-Window and have access to a Web browser such as Netscape Communicator and a Web server such as Apache. Using UNIX, you can also test your Web pages using the localhost feature.

## REVIEW QUESTIONS

1. True or False: HTML tags are special codes enclosed in square brackets ([]).
2. True or False: Most tags are used in pairs: one marks the beginning of a section while a corresponding tag marks the end of a section.
3. Which section of an HTML document contains the title?
   a. HEAD
   b. BODY
   c. PARAGRAPH
   d. BLOCKQUOTE
4. Text between the _____ tags appears centered.
   a. [CENTER] and [/CENTER]
   b. <CENTER> and </CENTER>
   c. <ALIGN=center> and </ALIGN>
   d. <JUSTIFY=center> and </JUSTIFY>
5. To run Netscape in UNIX or Linux, you _____.
   a. need not be in X-Window
   b. should be in X-Window
   c. must be in X-Window
   d. must have the Apache server installed
6. A basic HTML document has a _____.
   a. head and a body
   b. title and a body
   c. set of tags
   d. link to other Web pages

7. The _____ button in an HTML script is used to transfer the data to a Web server that passes it to a CGI script.
   a. Transfer
   b. Accept
   c. Go
   d. Submit
8. Which of the following is not true?
   a. Perl is the most commonly used CGI programming language.
   b. Perl is the only language used for CGI programming.
   c. CGI scripts do not require any language other than .cgi scripts.
   d. Perl CGI scripts are invoked by Web pages.
9. In Perl, the _____ character precedes an array variable name.
   a. $
   b. #
   c. %
   d. @

## EXERCISES

1. Create a personal Web page with your name, address, telephone number, and a brief paragraph describing your hobbies and interests. Your name should be centered in a large heading.
2. Design a Web page that allows the user to enter his or her age. The page should have a Submit button that, when clicked, invokes a CGI script. The script should display the user's age in days. (Don't worry about leap years.)
3. Design a Web page that allows the user to enter the width and length of a rectangle. The page should have a Submit button that, when clicked, invokes a CGI script. The script should display the area of the rectangle (width × length).

## DISCOVERY EXERCISES

1. Create an HTML and CGI Perl script to create an interactive Web page to accept your first and last name in the HTML file and pass it to the CGI Perl script, where it displays, "Hello," followed by your first and last name.
2. Create a Perl program to read both the students file and the number list file. Display the students' names in alphabetical order. Display the numbers in the order in which they were read.
3. Design a Web page that is a simple addition calculator. It should allow the user to enter two values. When the user clicks a Submit button, a CGI script returns the sum of the two numbers.